Vidi Language Reference

@davidberneda v0.0.13-alpha February-2021

https://github.com/davidberneda/Vidi

Important:
DRAFT. EVERYTHING MIGHT CHANGE.

General concepts

Vidi is a strict typing, object oriented language that *borrows* most of its features from existing languages like Java, C#, Delphi / Pascal and others.

It is *case-insensitive* by default. That means for example `Game` is considered equal to `gAmE`, but it can also be set to be case-sensitive.

Comments in code

The following examples contain *comments* as a single-line of text beginning with: `//`

Basic data types

Numeric

Numbers can be expressed in several ways:

```
123     // Integer
-4567   // Negative
12.345  // Float

4e2   // Exponent

// Other bases:

0xFF    // Hexadecimal base 16
0b11011 // Binary base 2
0c217   // Octal base 8
```

Text

Double or single quotes can be used to delimit text.

```
"Hello"        // Double-quotes
'World'        // Single-quotes
"How're you?"  // Single quote inside double quotes
'Say "abc" !'  // Double quotes inside single quotes
```

Booleans

```
True
False
```

Other data types

Arrays

```
[ 1, 2, 3 ]  // Simple array
[ ["a","b"], ["c", "d"] ]  // Array inside array
```

Integer Ranges

Ranges express minimum and maximum values:

```
1..10  // from 1 to 10
-12..-2
```

Ranges can be used in several places, like when declaring an array:

```
MyArray : Integer[1..10]
```

Or to specify custom `Integer` types to benefit from overflow checking:

```
// A custom Integer class
Podium is 1..3 {}

P : Podium := 4  // <-- Error. Overflow

// The 'Podium' class can also be used as an array dimension:

MyArray : Integer[Podium]  // same as Integer[1..3]
```

Or to use a range in a `for` loop:

`for Num in 0..1000 { }`

Or in function parameters and result types:

`MyFunction( MyParam : 20..1000): 4..10 { }`

Ranges can also be returned from functions:

```
Months : Range { 1..12 }
```

A range can also be used as a type of an array:

```
Podiums is 1..3[10] {}  // An array of 10 integer values, each value from 1 to 3
```

Expressions

Logical

Boolean operators:

`and or not xor`

Conditional operator:

`2 > 1 ? True : False   //  Ternary`

Membership operator:

```
'A' in 'ABC'  // True
5 in [1,2,3]  // False
```

Arithmetic

```
2 + 3 - 5 * (6 / -7) // Basic math

255 or 0xFF
128 and 255
64 xor 32
not 123

"Hello" + "World" // Text addition

// Other mathematical expressions using classes instead of symbols:

Math.Power(5,2)    // 5 elevated to 2 is: 25
Math.Modulo(10,3) // 10 modulo 3 is: 1

BinaryShift.Left(2,4)    // 2 << 4 is: 32
BinaryShift.Right(32768,4)  // 32768 >> 4 is: 2048
```

Comparative

Equality operators:

`=  <>  >  <  >=  <=`

Grouping

Parenthesis are used to group expressions and indicate precedence:

`(4+2) * 6 - (5/9) * (Abc - Xyz)`

Identifiers

Identifiers begin with an alpha character (`a` to `z`) or `_` (underline), and then any digit (`0` to `9`), alpha or underline.

Examples:

```
Abc
X123
My_Name
_Test4
_4Z
```

Variables

The `:` symbol (colon) is used to separate the variable identifier (variable name) and its type:

Simple variables:

```
A : Integer
B : Text
```

A variable can optionally define a *default* value (initial value) using the `:=` symbol:

`F : Float := 123.45   // Value initialization`

Variable type can be optionally omitted to infer it from its initial value:

```
Data ::= True      // Type inference. (Data is Boolean)
Planet ::= Earth   // Planet variable is of the same type as Earth value
```

Arrays are declared using the `[]` bracket symbols, and can also be optionally initialized:

```
Colors : Text[] := [ "Red", "Blue" ]

Matrix : Float[ 3,3 ]   // Alternative way: Float[3][3]
```

Ranges and expressions can also be used to declare array dimensions:

```
Numbers : Integer[ 1..(2*10) ]   // 20 elements, from 1 to 20
```

Multiple variables of the same type can be declared in a single line:

```
Name, Surname, Address : Text   // Three Text variables

// Also supported optional same value for multiple variables:
X,Y,Z : Float := 1.23

// Multiple variables type can be inferred:
A,B ::= True  // Both A and B are of Boolean type
```

Assignments

The `:=` symbol assigns (sets) the right-side value or expression, to the left-side variable:

```
X:Text
X := 'Hello'  // <-- assignment
```

Arithmetic assignments (`+= -= *= /=`) are supported:

```
Z : Integer

Z := 1

Z += 3  // Z := Z+3
Z -= 2  // Z := Z-2
Z *= 5  // Z := Z*5
Z /= 4  // Z := Z/4
```

Text and arrays support additions:

```
Hi : Text
Hi := 'Hello'
Hi += ' World!'  // string concatenation

Nums : Integer[]
Nums += [1,2,3]  // Array Nums.Append
Nums += 4
```

Copying and referencing

These data types (numeric, text and booleans) are "*value types*". They are always copied when assigning variables:

```
A : Integer := 123
B : Integer := A

// A and B are independent. Modifying A does not change B.
```

The rest of types (objects, arrays and functions) are always assigned by reference.

```
A : Person
B : Person := A

// A and B point to the same Person variable.
// Modifying one, changes the other.
```

Constants

The `final` keyword is used to define variables that cannot be modified (*readonly*):

```
final Pi : :=  3.1415
final Hello : Text := 'Hello'
```

Expressions are allowed to initialize final variables, including calling type-level functions:

```
final A ::= 1
final B ::= A + 1
final C ::= Math.Square(5) // 5*5 = 25
```

Classes

Structures, records, classes and interfaces are the same thing in Vidi.

```
Person {
  Name : Text
}
```

Class inheritance

A class can be extended from another class using the `is` keyword:

```
Customer is Person {
  Code : Integer
}
```

In the above example, the `Customer` class derives from the `Person` class.
`Person` is the ancestor class of `Customer`.

Everything is a class

The `sys` module contains most basic classes. The `SomeThing` class is the root of any other class.

Literal numbers, texts, arrays, etc are also classes. Types and routines are classes too. Everything is `SomeThing`.

```
SomeThing {}
```

Class as parameter

The `Self` keyword (equivalent to *this* or *it* or *base* in other languages) represents the class instance itself.

```
Foo is Integer {
  Bar() {
    SomeClass.Test(Self)  // Passing ourselves as a parameter to Test function
  }
}
```

Sub-classes and sub-methods

Class types and procedures / routines / methods / functions can be nested, unlimited, at any scope.

```
Life {      // class
  Tree {    // subclass

    Plant( Quantity : Integer) {  // method

      Forest is Text[] {    // subclass inside method
      }

      MyForest : Forest   // variable of Plant class

      SubMethod() { }
    }
  }
}
```

Sub elements are accessed using the `.` symbol, for example to declare a variable of a sub-class type:

`Pine : Life.Tree`

Class parameters

Exactly like methods, class parameters can be used when variables are declared, to initialize (construct) them.

```
Customer(SomeName:Text) is Person {
  Name:= SomeName
}

Cust1 : Customer("John")
Cust2 : Customer("Anne")
```

Generic types

There is no special syntax for generic types.
Class parameters of type `Type` can be used to specialize generic classes.

```
with Types

List(T:Type) is T[] {}    // Parameter of type: Type

Numbers is List(Float) {}  // List of Float
```

Casting expressions

As there are no pointers, casting is only allowed within types of the same class hierarchy.

```
Class1 {}
Class2 is Class1 {}

C2 : Class2
C1 : Class1 := C2  // OK

//  C2_bis : Class2 := C1 // <-- Error

C2_bis : Class2 := Class2(C1) // <-- Casting is OK
```

Automatic casting protection

Note: Experimental, not yet finished

```
MyBaseClass {}
MyDerivedClass is MyBaseClass { Foo : Integer }

MyDerivedData : MyDerivedClass
MyData : MyBaseClass := MyDerivedData

if MyData is MyDerivedClass
   MyData.Foo := 123   // allowed and safe because the above "if" makes casting
unnecessary

MyData.Foo := 456 // forbidden, compiler error. Casting is necessary.
```

Methods

Also called *routines*, *procedures* or *functions*.

`Area : Float { return 123 }`

Parameters

All parameters to a method are passed by default as read-only constants and cannot be modified.

```
Make( Wheels : Integer ) {
 // Wheels parameter cannot be changed
}
```

The `out` keyword in front of a parameter means the parameter must be assigned a value:

```
Parts( Style:Text, out Price:Float ):Boolean {
  Price:=123  // <-- Price must be assigned
}
```

The "Tuples" concept (returning more than one value) is done using records:

```
Format { Size:Integer Name:Text }  // <-- The tuple

// Routine returning the tuple:
MyFunction:Format {
  Result:Format
  Result.Size := 123
  Result.Name := 'abc'

  return result

  // Future releases might allow:  return 123, 'abc'
}

// Calling the routine and obtaining the tuple X:
X ::= MyFunction
```

Unnamed class types

Variables can also be declared using a class declaration just after the `:` colon symbol:

```
Planet : { Name:Text, Radius:Float }
Planet.Name := 'Saturn'

// An array can be used to initialize all class fields, in order:

Planet:= [ 'Saturn', 58232 ]

// Also array of arrays:

Planets : { Name:Text, Radius:Float } [] :=
  [
    [ 'Mars',   3389.5 ],
    [ 'Earth', 6371.0 ]
  ]
```

Many-Values parameters

The last parameter of a method can be declared with the special `...` prefix, to allow passing an undetermined number of parameters.

This is just syntactic sugar of passing an array without the need of typing the `[ ]` symbols around values.

```
Print( Values : Data...) {
 for Value in Values Console.PutLine(Value)
}
```

```
// Call examples:
Print
Print('abc')
Print(123,'abc',True)

TypedPrint( Values : Integer... ) {
 for Value in Values Console.PutLine(Value)
}

TypedPrint(7,8,9,10,11)  // similar to: [7,8,9,10,11]
```

Method Overloads

Routines can have the same name if they have different parameters and/or return values:

```
Write( Number : Integer) {}
Write( Number : Float):Integer[] {}
Write( Number : Text, Other : Boolean) {}
```

Method inheritance

A child class can declare methods with exactly the same name, parameters and return values as its ancestor parent class.

```
Class1 {
  Proc() {}
}

Class2 is Class1 {
  Proc() {
    Ancestor  // calls Class1.Proc
  }
}
```

The `Ancestor` keyword refers to its parent method.

Non-inheritable methods (final)

Methods can be declared `final` to forbid overriding them in derived classes.

`final Proc() {}`

Abstract methods

When a method body is empty, it is considered abstract.
There is no special syntax to declare it.

```
Test {
    Foo(A:Integer):Text {}  // <-- abstract method
}
```

That means the method cannot be called (it is an error at compile time), and that derived classes must implement (override) it and fill it with content.

Interfaces

There is no special syntax to declare interfaces.
Simple classes that have no fields (no variables), and all their methods are abstract, are always considered interfaces.

```
MyInterface {
   MyMethod( Data : Boolean ):Text {}   // abstract function
}
```

Classes can be derived from interfaces as usually:

```
MyClass is MyInterface {   // Deriving from an interface
  MyMethod( Data : Boolean ):Text { return "abc" } // must implement abstract method
}
```

Soft Interfaces

Classes that have methods with exactly the same name, parameters and return values of methods of an interface, can be used like instances of that interface.

```
SomeClass {
  MyMethod( Data : Boolean ):Text { return "abc" }
}

// This method requires a MyInterface parameter
Example( Value : MyInterface) {
  Value.MyMethod(True)
}

Some1 : SomeClass
Example(Some1)  // Some1 variable can be considered of MyInterface type
```

In the above code, `SomeClass` class is not derived from `MyInterface` but can be used as if it was.

Modules

The `with` keyword imports (loads) modules located in separate files.

It can be used anywhere on a file, not only at the top.
Imported symbols are only available at the scope after `with`.

```
with Module1, Module2, Module3.MyClass

MyClass {
  with SomeModule  // inner scope with

  Test : SomeClass // Class declared inside SomeModule
}

// SomeModule symbols cannot be accessed here, outside MyClass scope
```

Module file names might contain spaces or characters not allowed in module identifiers. In this case, the `with` syntax allows enclosing the module name in quotes like a text string literal:

```
with "My module with spaces"
```

Aliasing allows replacing module names with custom ones:

```
with Foo:= My_Long_Module.My_Class   // use Foo as alias

Foo1 : Foo  // variable of type My_Class
```

Grouping modules

A module can aggregate several other modules in one single place:

```
// use modules with aliases
with M1:=Module1, M2:=Module2  // etc

// Declare modules as new classes
Module1 is M1 {}
Module2 is M2 {}
```

When using the above module, the `Module1` and `Module2` contents will be ready available without needing to use them in `with` keywords.

Element visibility

The `hidden` keyword prefixing a class, field or method makes it unavailable outside its scope.

```
hidden MyClass {
  hidden MyField : Integer
  hidden MyFunction : Boolean {}
  hidden MySubClass {}
}
```

Unused hidden items will produce an error at compile-time.

Type-level shared elements

The `shared` keyword means an element (variable or method) belongs to type-level, not instance-level.
This is the equivalent of *class variables* in other languages.

```
Colors {
  shared Default : Text := "Red"
}

Colors.Default := "Blue"   // Can be used at type-level, without any instance
```

Type-level methods are auto-discovered. There is no special syntax to declare them.
When a method do not access any non-shared field or non-type level methods, it is considered `shared`.

```
// Type-level procedure, no shared keyword necessary
SetDefault( Value : Text) { Default:=Value }

Colors.SetDefault( "Green" )
```

Namespaces

There is no special syntax to declare namespaces.
Classes with no fields and no methods are considered namespaces.

```
// Module1
MyNamespace {
  MyClass {}
}
```

Modules with duplicate namespace names can be merged, to aggregate (contribute) new classes to the same namespace

```
// Module2
MyNamespace {
  OtherClass {}
}
```

The `with` keyword can also be used to reference just only a sub element instead of to everything in the module

```
// Module3
with Module1.MyNameSpace,
     Module2.MyNameSpace

Test is OtherClass {
  Some : MyClass
}
```

Strong Typing

Deriving one type from another just for the convenience of strict type checking:

```
// Type alias

Year is Integer {}
Y : Year

Month is Integer {}
M : Month

Y := M  // <-- Error. Different types.

Class1 {}
Class2 is Class1 {}

C1 : Class1
C2 : Class2 // := C1  <-- equivalent but forbidden (strict check)
```

Type discovery and reflection

The `Type` class provides methods to inspect (reflect) existing types:

```
if Type.is( C1, Class1 ) ...
Methods:Method[] := Type.Methods( C1 )
```

Extenders

At any scope, including in other modules, types can be extended with new methods.
So for example we can declare this class:

```
// Module 1
MyClass {
}
```

And then, inside the same module or in other external modules, we can declare new methods and subclasses of `MyClass` :

```
// Module 2
with Module1

MyClass.MyProcedure() {}    // New extended Procedure
MyClass.MySubClass { X:Float }  // New extended Sub-class
```

These new extended elements can then be used as normal, also in different modules

```
// Module 3
with Module1, Module2

Foo : MyClass
Foo.MyProcedure()   // Calling an extension as if it was a normal method

Bar : MyClass.MySubClass
Bar.X := 123
```

These extensions are only available inside the scope where they are declared.

Extended types can also be extended:

```
MyClass.MySubClass.MyNewMethod() {}
Bar.MyNewMethod()
```

Function types

A type can be used as a function declaration:

`MyProcType is (A:Text, B:Integer):Float {}`

This type can then be used anywhere like normal types:

```
Foo(Function:MyProcType) { Function('Hello',123) }
```

Functions can be signature-compatible with the custom type:

```
MyFunction(A:Text, B:Integer):Float {
  Console.PutLine(A, ' ', B.AsText)
}
```

The `MyFunction` variable (of type function), can now be called or passed to other methods.

```
Foo(MyFunction)  // shows 'Hello 123'
```

Anonymous functions

Also called *lambdas* or *callbacks* in other languages, functions can be passed as parameters "inline":

```
// Same as the above example "MyFunction", but unnamed
Foo(
 (A:Text, B:Integer):Float { Console.PutLine(A,' ',B) }
)
```

Or assigned to variables of the custom function type:

```
// Same as above, but using a variable
MyFunction_Variable : MyProcType := { Console.PutLine(A, ' ', B) }

Foo(MyFunction_Variable)
```

Enumerable types

The `is {}` syntax is used to declare enumerations.

`Colors is { Red, Green, Blue, Yellow }`

Variables and constants can then use the enumeration items:

`MyColor : := Colors.Blue`

These enumerations can also be used as dimensions for arrays:

```
Names : Text[Colors]  // Array of four text items
Names[Colors.Green] := "I Like Green"
```

And the `for in` statement can loop all the enumeration items:

```
for Color in Colors {
  Console.PutLine(Color)
}
```

Statements

Assignment

```
a := b
b := c + d
```

If

```
if a=b
   foo
else
   bar
```

While

```
while a=b {
  if a=0 break else a:= a - 1
}
```

Repeat

```
repeat {
  b += 1

  if b=5 continue

} until a<>b

// Single statements do not require { }

repeat
  b +=1
until b>5
```

For

A simple loop without any counter variable:

```
for 1 to 10 {}  // ten times
for 5..7 {}   // three times
```

An integer range:

```
for t in 1..10 {}   // ten times
```

Traditional loop using the `to` keyword:

```
a ::= 5    b ::= 7
for x:=a to b {}   // three times
```

The optional counter variable cannot be reused or accessed outside the `for` block.
It cannot be an already declared variable. Its type is inferred.

The `in` keyword can loop an enumerated type:

`for c in Colors {}`

The `in` keyword can also be used to loop an array:

```
Nums::=[ 6,2,9 ]
for i in Nums { Output.Write(i) }    // iterate an array
```

The array can be declared inline, without using a variable:

```
for i in [ 6,2,9 ] // the type of "i" is automatically inferred
```

A `Text` expression is an array of characters so it can also be iterated:

```
for c in "abc" {} // for each character
```

When

Also called *switch*, *select* or *case* in other languages.

```
Name::="Jane"

when Name {
  "Jane" { }
  "Peter" { }
}
```

Comparison complex expressions can also be used:

```
num ::= 5 abc ::= 3

when abc+num {
  <3 { Console.PutLine('Lower than 3') }
   4 { Console.PutLine('Equals 4') }
 <>6 { Console.PutLine('Different than 6') }
else { } // otherwise
}
```

After the first condition that matches the expression is found, execution flow exits.

Return

The return statement exits a method, with an optional value if the method is a function

```
Test {
  Foo() { return }
  Bar:Text { return "abc" }
}

// The return keyword is optional at the last expression of a function:
Square(X:Float) { X*X }
```

Try Catch

Error handling (exceptions) follows the standard of other languages using `try`, `catch`, `finally` keywords.

Code inside the `try` block is protected, so in case an error happens, the `finally` block of code is always executed:

```
try {
  X::=1/0   // <-- error divide by zero !
}
finally {
  Console.PutLine('Always executed')
}
```

The `catch` code is executed when a runtime error happens inside the `try` block:

```
try { X::=1/0 }
catch {
  // Optional code, might be empty {}
  Console.PutLine('An error happened')
}
```

The `catch` keyword can optionally specify a type (`MyError` class in this example), so only these errors will be processed:

```
MyError { Code:Integer }
Foo() { Exception.Raise(MyError) }  // <-- just an example of generating an
error

try { Foo() }
catch MyError {
  Console.PutLine('MyError happened')
}
```

If the `catch` type is prefixed with a variable name (`X` in the following example), then the fields of the error type can be accessed:

```
try { Foo() }
catch X:MyError {
  Console.PutLine('MyError happened: ', X.Code)
}
```

The `catch` and `finally` sections can coexist (`finally` must go after `catch` for clarity):

```
try { Foo() }
catch { Console.PutLine('Error happened') }
finally { Console.PutLine('Always executed') }
```

Multiple `catch` blocks can be used to respond to different errors. Duplicates are not allowed.

```
try { Foo() }
catch X:MyError { Console.PutLine('MyError happened: ', X.Code) }
catch DivideByZero { Console.PutLine('Division by zero') }
// catch DivideByZero {} <-- duplicate compile-time error
```

Recursivity

Methods can call themselves in a recursive way:

```
Factorial(x:Integer):Float {
  x=0 ? 1 : x * Factorial(x-1)
}

Factorial(5)  // Returns 120
```

Forward declarations

There are situations where methods should be called but are not yet declared.
These are handled automatically, no special syntax is necessary. (Note: Not yet developed)

```
TestInner(Work:Boolean) {} // <-- empty placeholder

TestForward() {
    TestInner(False)  // <-- not yet declared
}

// This replaces the placeholder above:
TestInner(Work:Boolean) {
  if Work
     TestForward
}

TestInner(True)
```

Properties

No special syntax for properties.
A property "getter" can be a field:

`Foo : Integer := 123`

Or a function:

`Foo : Integer { return MyFoo }`

And optionally, a property "setter" which is just a function with the same name:

`Foo(Value:Integer) { MyFoo:=Value } // Setter`

The compiler will handle property access transparently:

`Foo:=123  // will call the setter method: Foo(123)`

Finalizers

Classes can define a single, unnamed, parameter-less `final` method that will be called when variables get out of scope.

```
Shop {
  Console.PutLine( 'Open!' )

  final {
     Console.PutLine( 'Closed!' )
   }
}

{
  MyShop : Shop
  // do something with MyShop ...
} // <-- at the end of the scope, MyShop finalizer is called
```

Expression Operators

Note: Experimental, not yet finished

New expression operators can be implemented to provide cosmetic "syntax sugar" using symbols or keywords.

```
// Declare a new "is" operator, using the existing Type.is function
Operator.is := Type.is

// Use example
Foo : Boolean

// Equivalent expressions
if Type.is(Foo, Boolean) Console.PutLine('Ok')
if Foo is Boolean Console.PutLine('Ok')

// Existing basic operators like +, -, *, >, < etc could theoretically be re-implemented as extensions.
// The compiler finds the best overload for left and right types, if there is more than one.

A + B   // calls Integer.Add(A,B) if both A and B are Integer-compatible
```

Syntax

Reserved words

```
ancestor
and
break
catch
continue
else
False
final
finally
```

```
for
hidden
if
in
indexed
is
not
or
out
repeat
return
self
shared
to
True
try
until
when
while
xor
with
```

Reserved symbols

```
{ }   // code block
.     // membership  Foo.Bar
[ ]   // arrays   [1,2,3]
:=    // assignment Foo:=123
:     // type declaration Foo:Integer
,     // parameters 1,2,3
( )   // expression groups, parameters
..    // ranges 1..100
...   // many values parameter
?     // condition expression A=B ? 1 : 2
>     // greater than
>=    // greater or equal than
<     // lower than
<=    // lower or equal than
<>    // different than
+     // addition
-     // subtraction
*     // multiplication
/     // division
```

Comments

```
// Single line comments

/*
  Multiple line
  comments
*/

Inline : Text := "allow" + /* comments */ "around code"
```